Technical white paper

Concepts of NonStop SQL/MX

Part 5. Introduction to SQL/MX Stored Procedures

Table of contents

Introduction 2
Intended audience 2
Overview 2
Architecture of NonStop SPJ 2
Definition of Stored Procedures 2
Invocation of Stored Procedures 3
Examples of applications using SPJs 5
SPJs that use SQL/MX data 5
A special database connection for SPJs 5
Managing database connections 5
SPJ calling other Stored Procedures 6
Setting the run-time environment for MXUDR 6
Using UDR_JAVA_OPTIONS 6
Contents of UDR_JAVA_OPTIONS 6
Example UDR_JAVA_OPTIONS 8
Using MFC with SPJs 9
Enabling MFC 9
MFC settings are not automatically propagated 10
Propagating SQL environment settings to Module File Cache 10
Logging system for SPJs 11
Enable simple logging 11
Logging debug statements 12
Handling and reporting exceptions 13
Conclusion 13
References 13
Other articles in this series 14
Other interesting reads 14
Appendix: A sample SPJ 15

Click here to verify the latest version of this document

Introduction

This document contains condensed usage information for development and deployment of Stored Procedures (SP) for NonStop SQL/MX. This brief is an addition to, not a replacement, of the *HP NonStop SQL/MX Guide to Stored Procedures in Java*.

This document refers to use of Apache log4j; the documentation for log4j can be found at logging.apache.org/log4j/1.2/index.html.

Intended audience

This paper is written for people who know how to use Stored Procedures but may not be aware of certain important run-time details that apply to NonStop SQL/MX.

Overview

Because Stored Procedures for NonStop SQL/MX are written in the Java language, they are often referred to as SPJs (Stored Procedures in Java).

SPJs are executed "close to the database," just as they are in other DBMSes. The execution environment is a separate process that runs on the NonStop server. This process, called the MXUDR[1], is a process that acts as a wrapper around a Java Virtual Machine (JVM). The JVM in turn loads and executes the SPJs. MXUDR processes are started automatically by NonStop SQL/MX when needed.

SPJs for NonStop SQL/MX can use all the features that Java supports in a headless environment, such as sending email and accessing other NonStop services. The JVM loads an SPJ class based on its location as defined in the SQL/MX catalog. If any external Java classes are used, they can be defined in the run-time CLASSPATH of the MXUDR. This is described later in the section called Setting the run-time environment for MXUDR.

Stored procedures do not have to access the NonStop SQL/MX database, but in most cases they will. This document discusses SPJs that access the database.

Architecture of NonStop SPJ

This section provides a brief overview of the implementation. There are two important aspects: the definition of the Stored Procedures and their invocation at execution time.

Definition of Stored Procedures

Stored Procedures are defined using SQL syntax, and their definition data is stored in the SQL/MX metadata. Specific data for SPJs is stored in a metadata table called ROUTINES[2].

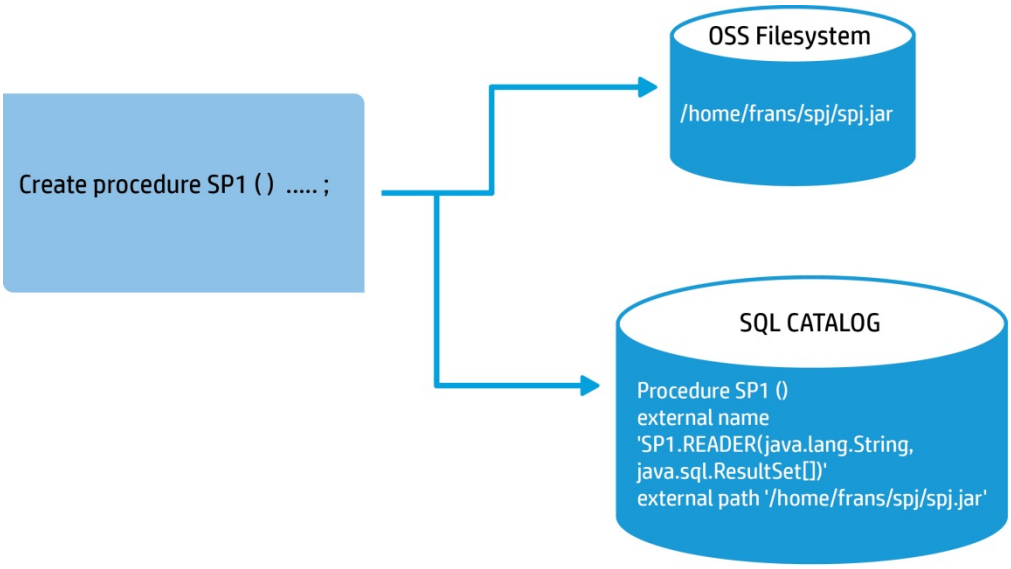
Figure 1. Definition of an SPJ in SQL

```
create procedure SP1Reader( IN tableName varchar(255) )
dynamic result sets 1
external name 'SP1Reader.Reader(java.lang.String, java.sql.ResultSet[])'
external path '/home/frans/spj/spj.jar'
reads SQL DATA
language java
parameter style java;
```

The definition in the metadata contains all the attributes the SQL executor requires to setup an invocation of the procedure and to return results to the caller, but not the SP code itself. SPJs can be developed and compiled using a Java development environment such as Eclipse. The compiled Java code of the SP must be placed in the NonStop Server's OSS file system as simple class files or in Java Archive (jar) files.

[1] MXUDR is short for MX User Defined Routine. The object name is /G/system/system/mxudr.
[2] The metadata view ZZPROCEDURES provides this data in a user-friendly format. Metadata views are described in the paper Introduction to SQL/MX Metadata.

Figure 2. Definition flow of an SPJ



When a Stored Procedure is defined, the definition is validated against the Java code that is stored at the "external_path" location. Figure 2 shows how "Create procedure SP1" validates the existence of the object that is specified the "external path" attribute. If the class exists, the method signature is validated against the Java signature that is specified in "external name".

Invocation of Stored Procedures

Stored Procedures are invoked by an application using the SQL "CALL" API. They execute in a separate process that is dedicated to the application process[3] that issues that CALL. This process is called the MX User Defined Routine (MXUDR). It hosts a Java Virtual Machine (JVM) which in turn provides the execution environment for the SPJ. The JVM is responsible for loading and executing the Java code.

The MXUDR process starts automatically when the SQL/MX executor needs to execute the first CALL statement. It will remain active as long as the process that launched it remains active, thus acting as the environment for all the SPJs for that application process. In most cases, there will be only one MXUDR process for a given calling process, but this does not have to be the case, as we will see later in the Setting the run-time environment for MXUDR section of this paper.

When an application prepares a SQL CALL statement to invoke the Stored Procedure, the SQL/MX compiler retrieves the necessary metadata from catalog tables and creates an execution plan. When the prepared CALL statement is executed, as part of the execution plan, the SQL executor library in the application and the MXUDR instance communicate via SQL/MX-specific inter-process messages, and the MXUDR wrapper code passes the data to the SP that executes within the JVM.

[3] Note that ODBC or Java programs using the T4 driver communicate via the MXCS subsystem in which case the application process is represented by a server process of MXCS.

Figure 3. Application invoking SPJs

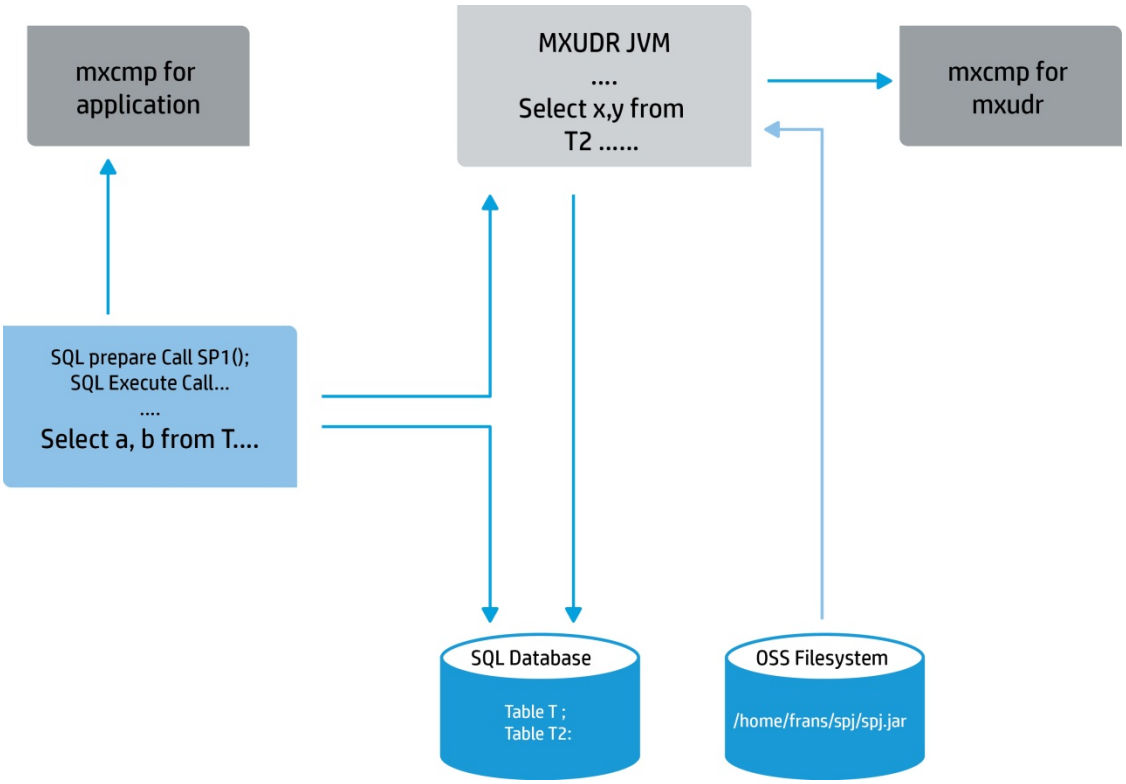


Figure 3 shows the following flow of events:

- The application prepares a SQL call to a stored procedure SP1 [4]
  - The mxcmp instance for the application reads the SQL/MX catalog metadata (not shown in the diagram) and creates the execution plan
- The application executes the SQL call
  - The executor within the application executes the plan and sends the request to the MXUDR process that contains the JVM
  - The JVM loads the SP class from the OSS file system and executes the Java code
- The SQL statements that the SP executes are compiled by the mxcmp instance that belongs to the MXUDR instance
- The result set from the query is returned to the caller
- The diagram also shows the execution of a regular SQL select, which is handled by the executor within the application process. Its execution plan will also be generated by the mxcmp instance for the application.

[4] This example shows the flow of events for an application using dynamic SQL. An application that uses embedded SQL will not prepare the statement at run time but as a separate step prior to application execution.

Examples of applications using SPJs

When an application calls a Stored Procedure, the executor creates an MXUDR instance which in turn launches an instance of the SQL/MX compiler, mxcmp. For a system manager, it may not always be obvious why application processes start MXUDR instances. The following table shows applications and why they might communicate with MXUDR processes.

Process	Event
mxci command interface	CALL <stored procedure> statement issued
mxosrvr process	Associated ODBC or JDBC T4 client issued CALL statement
JVM process	Java application using T2 driver issued CALL statement
COBOL/C/C++ server	CALL statement issued
MXUDR process	Stored procedure calling another Stored Procedure
Any application process	SQL Executor firing a trigger that invokes a Stored Procedure

Note
When a Stored Procedure calls another SPJ, a second MXUDR process will be started for that CALL. HP recommends that SPJs invoke those Java methods directly using the Java API instead of issuing another SQL CALL.

SPJs that use SQL/MX data

Although there is no requirement for an SPJ to access a SQL/MX database, most do. These SPJs are written just like any other Java class that uses SQL/MX. In order to use the database, there must be a database connection, SQL statements must be prepared and executed, exceptions must be handled, and so forth.

A special database connection for SPJs

In order to access a SQL/MX database from Java, a connection must be established. Stored Procedures use a standard connection URL, called **`jdbc:default:connection`**. This URL is also used in other implementations of Stored Procedures like Oracle, IBM DB2 and Apache Derby. Using this URL simplifies programming in the following ways:

- Loading a specific driver is not needed, because the SPJ runs within control of the DBMS, which knows what the driver should be. SPJs for NonStop SQL use the T2 driver.
- User / password credentials don't need to be specified.
- Connections are automatically closed when the caller closes the CALL statement.

Managing database connections

Connections are a scarce resource on most systems. On the NonStop server, every connection includes an instance of the SQL/MX compiler. Too many of these are wasteful, so customers may want to keep the number of connections low. Most SPJs use only a single connection, because an SPJ typically executes "waited" (the caller waits for the SPJ to finish), as a single thread[5].

Each time an SPJ executes, it creates a connection, prepares one or more SQL statements, executes them and returns results to the caller. When the calling process closes the SQL CALL statement, its executor closes the connection that was created by the SPJ. Failure to close the CALL statements leads to database connection leaking in the MXUDR process.

To detect leaking of connections, consider using a connection pool of just 1 for the SPJ as described later in Define JDBC pooling. When a statement is not closed properly, a second call will cause an exception to be raised ("*Attempting to exceed the maximum connection pool size (1)*").

[5] In principle, an SP could launch multiple threads. This is not recommended and is hardly useful, because the SQL/MX statements will be executed "waited", halting the JVM to wait for SQL to return. The JVM jdbcmx.sqlmx_nowait attribute is set to OFF when the MXUDR program is launched.

SPJ calling other Stored Procedures

Stored Procedures can use the SQL CALL statement to call other SPs. However, because the executor within the MXUDR process will launch another MXUDR to execute the second SPJ, it is better to invoke the implementation of the second SP as a normal Java method, passing it the connection object and other parameters. A good practice is to write an SP with two signatures, one with and one without a connection object.

Setting the run-time environment for MXUDR

The MXUDR process provides the execution environment for the Stored Procedures. The execution environment of the SPJ-or, better, the runtime environment of the MXUDR process- needs to be in place before any SP is called. Some parts of the runtime environment are:

- The Java runtime version that is to be used. If not explicitly defined, this is the default location of the Java runtime environment `/usr/tandem/java`
- The JDBC Driver location. If not explicitly defined, this is the default location of the JDBC/MX T2 driver, `/usr/tandem/jdbcMx/current/lib`
- The location of the classes that implement the SP (This is defined in the SQL/MX metadata when the SP was created and does not need to be passed explicitly.)
- The location of the classes that are invoked by the SP
- The catalog and schema that are the defaults for the SP (These are derived from the catalog definition and do not have to be specified.)
- Other global JVM settings such as enabling Module File Cache (MFC), statement and connection pool size, memory and logging

The mechanism to pass this information is a Control Query Default parameter called UDR_JAVA_OPTIONS.

Using UDR_JAVA_OPTIONS

When a process performs the first call to an SPJ, an MXUDR process is created on the NonStop server. Because it is an OSS process, it inherits the execution environment from the caller process like any other OSS process when it is forked by its parent. For example, when mxci is used to call an SPJ, it creates an MXUDR process which inherits any values for the CLASSPATH and library variables from the mxci shell environment. However, if an SPJ requires a different execution environment (different CLASSPATH and library variables), that environment needs to be explicitly set by mxci.

By contrast, when remote-mxci (rmxci) is used to call an SPJ, the request is actually executed by a server instance of the MX Communication Service subsystem (MXCS) Data Source that executes all SQL statements on behalf of rmxci. The execution environment is defined by MXCS unless it is explicitly set by rmxci.

SQL/MX uses the UDR_JAVA_OPTIONS CQD[6] to pass startup information to the MXUDR process. All CALL statements with the same value for UDR_JAVA_OPTIONS will be passed to the same MXUDR instance. This means that generally, there will be one single MXUDR instance per database connection of a calling process, but it is possible to create multiple instances if they require specific runtime settings. If an MXCS Data Source is used by many different user-logons, multiple instances of MXUDR will be created, one for each new user. Chapter 1 of the *SQL/MX Guide to Stored Procedures in Java explains* the use of multiple values of the CQD in more detail.

Contents of UDR_JAVA_OPTIONS

A program can set the value for UDR_JAVA_OPTIONS by issuing a SQL/MX Control Query Default statement. The syntax is:

```
CONTROL QUERY DEFAULT UDR_JAVA_OPTIONS  ' <option_string> [<option_string>  .....]';
```

where `option_string` is a system property for the JVM.

For example this string:

```
-Djdbcmx.maxStatements=100
```

[6] CQD is short for "Control Query Default, a means to set environment values for execution of SQL statements, overriding the values in the SYSTEM_DEFAULTS metadata table.

sets the JDBCMX in-memory statement cache to contain 100 entries.

CQDs can be set in the MXCS Data Source definition. This allows the system administrator to use specific settings for specific purposes. The mxosrvr process will set the CQD for each connection that is made. CQDs are defined in MXCS as environmental variables: EVAR, TYPE CONTROL. To define the maxPoolSize and and maxStatements properties for a Data Source called MYDS, the following setting is used:

```
ADD EVAR $MXOAS.MYDS.UDR_JAVA_OPTIONS, TYPE CONTROL ,

VALUE '-Djdbcmx.maxPoolSize=1 -Djdbcmx.maxStatements=100' ;
```

The TS/MP (Pathway) server class definition allows setting environment variables but does not support setting CQDs for SQL/MX. However, it is possible to set properties for a JVM by setting the environment variable _JAVA_OPTIONS[7] to contain the required Java properties, just as they are defined in UDR_JAVA_OPTIONS. For example, to set the same maxPoolSize and maxStatements properties as in the previous example, the following text can be added to the server definition:

```
ENV _JAVA_OPTIONS="-Djdbcmx.maxPoolSize=1 -Djdbcmx.maxStatements=100"
```

Finally, UDR_JAVA_OPTIONS can be set as a system-wide default by including a row in the SYSTEM_DEFAULTS metadata table. However, this "one-size-fits-all" setting may not actually fit all user applications.

The following sections provide an overview of the most appropriate options or properties. They do not have to be in any specific order. However, because certain properties belong together, the recommendation is to group them logically.

Controlling Java Runtime Environment (JRE)

This property tells MXUDR which JVM version is used to execute the SPJs. NonStop SQL/MX release 3.2.1 and higher require Java 7 to run SPJs. If the system default JVM version is less than 1.7, the `jrehome` property must be set explicitly as shown in the example.

```
-Dsqlmx.udr.jrehome=/usr/tandem/nssjava/jdk170_h70/jre
```

If not specified the JRE defaults to `/usr/tandem/java`. Other system properties, such as memory size definitions for the JVM, can also be included when setting the JRE.

Controlling the JDBC versions

SPJs use the JDBC T2 driver to access the database. By default, MXUDR searches for the driver in the JDBCMX default location `/usr/tandem/jdbcMx/current/lib`.

The T2 driver has a Java component and a native library component. These two components must match the same release. Therefore it is best to include both when the default location is not used.

```
-Dsqlmx.udr.extensions=/usr/tandem/jdbcMx/T1275R32/lib/jdbcMx.jar

-Djava.library.path=/usr/tandem/jdbcMx/T1275R32/lib/
```

The example shows two properties: `sqlmx.udr.extensions`, containing the Java component, and `java.library.path`, containing the native component of the driver. Note that the Java component includes the file name (jdbcMx.jar), but the native component refers to the directory that contains the DLL, not the DLL itself.

Define JDBC pooling

Connection pooling should be used to prevent connections from being physically closed and reopened every time. A pool size of 1 is sufficient, since SPs are serving one connection at a time. The maxStatements property sets the in-memory statement cache for prepared statements.

```
-Djdbcmx.maxPoolSize=1 -Djdbcmx.maxStatements=100
```

Access to additional classes

Additional classes, for example, utility classes that are present in archives elsewhere on the system, can be added to the `java.class.path` property.

```
-Djava.class.path=/home/frans/lib/log4j-1.2.16.jar:/home/frans/lib/MyUtils.jar
```

This example applies to an SPJ that uses log4j and user classes that are referenced by SPJs but are not present in the same jar as the SPJs.

[7] _JAVA_OPTIONS (note the leading underscore), is an environment variable that a JVM (on any platform) uses to set system properties that are normally included when the JVM is invoked.

Other properties of interest

The order of properties is not important; however it is best to maintain a logical grouping, starting with the ones explained above, followed by the optional settings described below.

Set Module File Caching

Module File Caching (MFC) is discussed later in this paper, in the section called Using MFC with SPJs.

```
-Djdbcmx.compiledModuleLocation=/home/frans/spj/T2modules -Djdbcmx.enableMFC=ON
```

Define JDBC tracing

SPJs use the T2 driver and therefore they require the T2 settings for JDBC tracing. The trace log file will be located on the NonStop server.

```
-Djdbcmx.traceFile=/home/frans/SPJlogs/ -Djdbcmx.traceFlag=3
```

To prevent multiple instances writing into the same file, a directory name should be specified as the traceFile. If the traceFile points to a directory name, the driver will generate trace files in that directory for each instance of the JVM, for example: `20130527-030650-777060452.log`. The first part of the pathname is the date, the second part is the create-time, and the third part is the OSS Process ID of the MXUDR process that creates the log.

Define log4j configuration

Apache log4j uses a default configuration file, called `log4.properties` or `log4j.xml`, and searches for it in the locations specified in the CLASSPATH of the MXUDR process. A good practice is to include a configuration file in the jar that also contains the SPJ. However, a more flexible method is to place the configuration file separately in the file system. This way, debugging can be facilitated by using a separate set of log4j properties for debugging as shown in the following example.

```
-Dlog4j.configuration=file:/home/frans/spj/log4j.xml
```
[8]

Further discussion of logging appears in the section Log4j logging with SPJs later in this paper.

Set debug options for debugging with Eclipse

Java programs running on the NonStop Server can be debugged remotely using the Eclipse development environment. Because the Stored Procedures run within the MXUDR process, which is a container for the JVM, SPJs can also be debugged remotely. To enable remote debugging, all of the following options need to be set:

```
-Xrunjdwp:server=y,transport=dt_socket,suspend=y,address=<free_server_TCPIP_port>

 -Xdebug -Xnoagent -Djava.compiler=none
```

Example UDR_JAVA_OPTIONS

The SQL/MX command interface, mxci, can execute scripted commands using the OBEY command. An mxci obey script provides an easy and flexible way to establish the environment before calling an SPJ.

The example shown in Figure 4 begins with the creation of an mxci session from an OSS prompt. After the startup banner display, the user commanded mxci to `obey setudr.sql`. The remainder of the display comprises echoed script commands and command output from mxci.

This example uses mxci variables, called patterns, in the script syntax to construct a CONTROL QUERY DEFAULT statement that is used to prepare the SPJ environment. The obey file sets five individual mxci patterns, the first one is set to contain a single quote. That pattern is used to provide the required single quotes when the CQD is set. Other “set pattern” commands are prefixed with the comment indicator (--), so they have no effect. The “control query default UDR_JAVA_OPTIONS” command concatenates all of the patterns, bracketed by single quotes. The output of the “showcontrol default” command maps to the four values set in the patterns.

[8] While a log4j.properties file is the easiest to use, many developers prefer using XML to configure log4j. The configuration is then stored in a file called log4j.xml.

Figure 4. Use of UDR_JAVA_OPTIONS in mxci

```
~/spj> mxci
Hewlett-Packard NonStop(TM) SQL/MX Conversational Interface 3.2.1
(c) Copyright 2003, 2004-2013 Hewlett-Packard Development Company, LP.
>>obey setudr.sql;
>>-- setting the UDR_JAVA_OPTIONS to use SPJ from mxci
>>-- the $$Q$$ variable will contain only a single quote
>>set pattern $$Q$$    '";
>>-- set pattern $$JRE$$   '-Dsqlmx.udr.jrehome=/usr/tandem/java/jre';
>>-- set pattern $$JDBCMX$$ '-Dsqlmx.udr.extensions=/usr/tandem/jdbcMx/current/lib/jdbcMx.jar -
Djava.library.path=/usr/tandem/jdbcMx/current/lib/ ';
>>-- set pattern $$JDBCMX$$ '-Dsqlmx.udr.extensions=/home/frans/lib/jdbcMx.jar -
Djava.library.path=/home/frans/lib/ ';
>>set pattern $$CP$$    '-Djava.class.path=/home/frans/lib/log4j-1.2.16.jar ';
>>-- set pattern $$MFC$$   '-Djdbcmx.compiledModuleLocation=/home/frans/modules/spj -Djdbcmx.enableMFC=ON' ;
>>set pattern $$POOL$$   '-Djdbcmx.maxPoolSize=1 -Djdbcmx.maxStatements=100';
>>set pattern $$LOG$$   '-Djdbcmx.traceFile=/home/frans/spj/jdbclogs -Djdbcmx.traceFlag=1';
>>set pattern $$LOG4J$$  '-Dlog4j.configuration=file:/home/frans/spj/log4j.xml';
>>
>>
>>control query default UDR_JAVA_OPTIONS
+>$$Q$$ $$JRE$$ $$JDBCMX$$ $$CP$$ $$MFC$$ $$POOL$$ $$LOG4J$$ $$LOG$$ $$Q$$ ;

--- SQL operation complete.
>>showcontrol default;

CONTROL QUERY DEFAULT
 UDR_JAVA_OPTIONS
   -Djava.class.path=/home/frans/lib/log4j-1.2.16.jar
   -Djdbcmx.maxPoolSize=1 -Djdbcmx.maxStatements=100
   -Dlog4j.configuration=file:/home/frans/spj/log4j.xml
   -Djdbcmx.traceFile=/home/frans/spj/jdbclogs -Djdbcmx.traceFlag=1

--- SQL operation complete.
>>
```

Using MFC with SPJs

Module File Caching (MFC) is a technique to reduce SQL compilations by creating a system wide query cache using SQL/MX module files which are files that contain compiled SQL execution plans. This makes MFC an ideal feature to be used with SPJs, because SPJs are expected to provide good performance.

In order to enable MFC for a JDBC T2 driver, two system properties must be passed to the MXUDR JVM. One, `jdbcmx.enableMFC=ON`, tells the driver to use MFC, and the other, `jdbcmx.compiledModuleLocation=<directory name>`, defines the location that contains the compiled modules and source files that are used to build the modules. The directory must exist when the JVM is started; the JVM will abort if the module directory does not exist.

Enabling MFC

The most common way to enable MFC for SPJs is to add the two properties above to the UDR_JAVA_OPTIONS settings. As an alternative, the SPJ itself can call an initialization routine that sets these properties before the first connection is created. This allows the SPJ to control MFC without having to rely on the caller to do so.

Implementing this alternative method in a flexible way requires a designated configuration file or SQL table. The example in Figure 5 uses a simple, fixed location to store the MFC modules.

Figure 5. Enabling MFC from within an SPJ

```
if (!initialized) {
// this could be configuration coming from a configuration table;
                                    System.setProperty("enableMFC", "ON" );
                                    System.setProperty("compiledModuleLocation",
                                    "/home/frans/spj/T2modules");

                                initialized = true;
                            }
```

MFC settings are not automatically propagated

Module File Caching can be used by application programs and by the SPJs they call. However, the MFC settings of the caller are not propagated to the Stored Procedures they call.

For example, if a JDBC T4 application uses a mix of SQL statements and calls to SPs, MFC needs to be enabled in both environments: the MXCS Data Source that is executing the statements on behalf of the T4 driver application *and* the MXUDR instances that will be started by MXCS processes when a CALL statement is executed.

The MXCS subsystem is configured using the following SET and CONTROL variables (called EVARs):

Figure 6. Enabling MFC in MXCS

```
-- Control settings for the MXCS Data Source --
STATEMENT_MODULE_CACHING , type SET, value 'TRUE';
COMPILED_MODULE_LOCATION, type SET, value '<module_directory>';

-- Control setting for the Stored Procedures
UDR_JAVA_OPTIONS, type CONTROL, value '<-DenableMFC=ON,
 -DcompiledModuleLocation=<module_directory> [ other options_strings] >'.
```

As indicated in the comment lines of the example, the SET variables are used to define MFC for the application that calls the Stored Procedure, while the CONTROL variable is used to define it for the SQL statements that the Stored Procedures execute.

To enable MFC for SPJs without enabling it on other statements, use only UDR_JAVA_OPTIONS. This may be useful in cases where the T4 application is using many unique SQL statements that cannot re-use any compiled modules.

Propagating SQL environment settings to Module File Cache

When the T2 driver detects that MFC is used and a new execution plan needs to be compiled, it creates a source file that contains the SQL statement plus all the active CQDs which make up the SQL environment. This ensures that the compiled execution plan contains all the necessary settings.

Note
Some SQL attributes, such as transaction isolation, which can be set by invoking a connection-specific method, may not be stored in a generated MFC plan. For this reason, use Control Query Default to set controls that influence query behavior. Remember, modules define a system-wide cache, not a connection-specific one. The next example illustrates this.

In Java, the following connection method invocation instructs the compiler to generate subsequent queries with READ_UNCOMMITTED access.

```
conn.setTransactionIsolation(Connection.TRANSACTION_READ_UNCOMMITTED);
```

However, the driver might find the statement in the module cache, and no new prepare will occur. To ensure READ_UNCOMMITTED access is included in the generated module, use the following statement that sets a CQD:

```
conn.createStatement().execute("control query default isolation_level 'read
uncommitted'");
```

The T2 driver will add CQDs that are dynamically set by the program to the generated source code. By contrast, when ODBC or the T4 driver is used, only the CQDs that are configured in MXCS will be included in the generated source code for the MFC modules. When specific CQDs are required in an execution plan, it is better not to rely on a driver to copy them, but to manually change the generated source code and regenerate the module.

Logging system for SPJs

Stored procedures run in their own environment (the MXUDR) and are not attached to a terminal. This means there is no convenient place to log events or debug messages during development. A simple message generated by `System.out.println("I have found an error!");` will not show up anywhere. This is why it is a good practice to set up a common logging system for SPJs.

There are two main logging systems for Java: Apache log4j and the java.util.logging package. Apache log4j is the most commonly used way to log events from Java. The examples shown here are based on log4j. The advantage of using this technique is that one can include debug statements in the programs, but the data they produce is controlled by configuration settings outside of these programs. Programs do not have to be changed to enable the debug statements. Detailed documentation for Apache log4j can be found at logging.apache.org/log4j/1.2/index.html.

Enable simple logging

When little data is logged, a configuration like the one shown below can be used:

Figure 7. Simple log4j configuration

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<!-- simple.xml: Simple log4j example      -->

<log4j:configuration xmlns:log4j='http://jakarta.apache.org/log4j/' >
<appender name="FA" class="org.apache.log4j.FileAppender">
<param name="File" value="/home/frans/spj/spj.log"/>
        <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="%d{dd-MMM HH:mm:ss} %-5p %c{2} %x - %m%n" />
        </layout>
</appender>

<root>
        <level value="INFO" />
        <appender-ref ref="FA" />
</root>
</log4j:configuration>
```

The following example shows output from an SPJ that displays the connection and the value of the JDBC trace file name once when it is initialized. Every statement is logged as an informational message, and SQL exceptions are logged as errors.

Figure 8. Simple log4j output

```
spj> cat spj.log
14-Jun 08:10:03 INFO  SPreader  - Tracing JDBC to /home/frans/spj/jdbcMx.log
14-Jun 08:10:03 INFO  SPreader  - using jdbc:default:connection
14-Jun 08:10:03 INFO  SPreader  - Statement: select * from zztables

15-Jun 06:24:24 INFO  SPreader  - Statement: select 1 from dual
15-Jun 06:24:24 ERROR SPreader  -  ----- SQLException -----
15-Jun 06:24:24 ERROR SPreader  -  SQL State:  42000
15-Jun 06:24:24 ERROR SPreader  -  Error Code: -15001
15-Jun 06:24:24 ERROR SPreader  -  Message:   *** ERROR[15001] A syntax error occurred at or before:
select 1 from dual;
   ^
15-Jun 06:24:24 ERROR SPreader  -  ----- SQLException -----
15-Jun 06:24:24 ERROR SPreader  -  SQL State:  X08MU
15-Jun 06:24:24 ERROR SPreader  -  Error Code: -8822
13-Jun 06:24:24 ERROR SPreader  -  Message:   *** ERROR[8822] The statement was not prepared.
....  .....

spj>
```

If the number of events is low, all SPJs can log output, such as error messages, to a single log file.

Logging debug statements

When specific debugging of an SPJ is required, a designated debug log file, separated from the other SPJ logs, should be used. In this example, the “debug session” uses a specific log4j configuration, and its name is included in the UDR_JAVA_OPTIONS string. The lines in bold indicate the changed lines from the simple configuration that was shown earlier.

Figure 9. Debug log4j configuration

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<!-- debug.xml  log4j example for debugging  -->

<log4j:configuration xmlns:log4j='http://jakarta.apache.org/log4j/' >
<appender name="FA" class="org.apache.log4j.FileAppender">
        <param name="File" value="/home/frans/spj/debug.log"/>
        <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="%d{dd-MMM HH:mm:ss} %-5p %c{2} %x - %m%n" />
        </layout>
</appender>

<root>
        <level value="DEBUG" />
        <appender-ref ref="FA" />
</root>
</log4j:configuration>
```

Note that it is also possible to include a specific debug logger for a single class in your configuration, for example:

```
<logger name="SP1Reader">
   <level value="DEBUG"/>
</logger>
```

Flexible naming of log4j output filenames can be achieved by using relevant system properties, like the NonStop process and logical CPU identifiers. This example uses the `nsk.process.cpu` property, which contains the logical CPU number in which the MXUDR process is running, and the `nsk.process.pin` property, which contains the process number within that CPU.

```
<param name="File" value="/home/hp/frans/spj/SPJ_${nsk.process.cpu}
_${nsk.process.pin}.log"/>
```

A process running in CPU 2 with process number 1125 will write a file called SPJ_2_1125.

Handling and reporting exceptions

In addition to logging for system and development purposes, an SPJ might need to provide error notifications to its calling program. Such notifications might be a user-friendly representation of a SQL exception, because information needed by an application developer or system administrator is rarely appropriate for an end-user.

It is good practice to combine the logging that was described earlier with communicating the error to the calling program in a meaningful way.

Designated error codes for Stored Procedures

SQL/MX reserves SQLSTATE values 38001 through 38999 to signal user-defined error conditions returned by Stored Procedures. This allows an SPJ to communicate error conditions which may or may not be related to SQL.

The following code snippet shows how the example SPJ included in the Appendix throws an exception upon request. When the string that normally contains a table name contains the value “throwit”, the method throws a new SQLException with an informative message, and the calling program will see a SQLSTATE value of 38088.

Figure 10. Using a designated error code and message

```
if (inString.equalsIgnoreCase("throwit") ) {
        logger.info("Throwing a test user Exception");
        throw new java.sql.SQLException("This exception is thrown upon request", "38088");
}
```

Conclusion

This paper described the architecture and definition of Stored Procedures in Java (SPJs) for NonStop SQL/MX. Applications that access SQL/MX databases can run on the NonStop Server or they can access the database from remote platforms using ODBC or JDBC T4 drivers. SPJs however, always execute in a run-time environment on the NonStop server. SQL/MX allows an application to pass configuration details to the run-time environment using a Control Query Default (CQD), called UDR_JAVA_OPTIONS. Module File Caching (MFC), an important feature to increase performance, is one of configuration details that can be set using this CQD.

SPJs can use the Apache log4j or java.util.logging packages to write exception or other information to a central location for later inspection. The paper described how to configure Apache log4j depending on the type of application (shell program, JDBC or ODBC, TS/MP servers) that invokes the SPJ.

References

More information can be found in the *HP NonStop SQL/MX Guide to Stored Procedures in Java*. Module File Caching (MFC) and the MX Communication Service subsystem (MXCS) are further described in the *SQL/MX Connectivity Service Administrative Command Reference manual*, the *SQL/MX Connectivity Service manual* and the *JDBC Driver manuals*.

The following NonStop SQL/MX manuals can be found in the *NonStop Technical Library* at: hp.com/go/nonstop-docs. From this page, select the appropriate server model, H-Series or J-Series, select ‘Manuals’ and then ‘User guide’. The manual titles below are the titles as they appear in the Technical Library without the release number, which is 3.2.1 at the time of this writing.

SQL/MX Guide to Stored Procedures in Java

SQL/MX Reference Manual

SQL/MX Installation and Management Guide

SQL/MX Connectivity Service Administrative Command Reference manual

SQL/MX Connectivity Service manual

JDBC Type 2 Driver Programmer's Reference for SQL/MX

JDBC Type 4 Driver Programmer's Reference for SQL/MX

Examples of Stored procedures can be downloaded from hp.com/go/softwaredepot; select *HP NonStop* followed by *Download HP NonStop online Help and Sample Programs.*

Documentation for log4j can be found at logging.apache.org/log4j/1.2/index.html.

Other articles in this series

Connecting and the initial environment	July 2011 (Part 1).
Introduction to Catalogs and other Objects	July 2011 (Part 2).
Introduction to SQL/MX Metadata	July 2011 (Part 3).
Introduction to SQL/MX Storage	May 2012 (Part 4).

Other interesting reads

A Comparison of NonStop SQL/MX and Oracle 11g RAC—Technical Whitepaper, HP document ID: 4AA3-2001ENW.pdf.

Module File Caching for NonStop SQL/MX—Technical Whitepaper, HP Document ID: 4AA3-8922ENW.pdf. (h20195.www2.hp.com/V2/GetDocument.aspx?docname=4AA3-8922ENW)

Appendix: A sample SPJ

The following is a very simple Stored Procedure (SP1Reader in the earlier examples). It shows the use of log4j and error reporting.

SP1Reader is defined in mxci as follows:

```
create procedure SP1Reader( IN tableName varchar(255) )
dynamic result sets 1
external name 'SP1Reader.Reader(java.lang.String, java.sql.ResultSet[])'
external path '/home/hp/frans/projects/spj/spj.jar'
reads SQL DATA
language java
parameter style java;
```

The following is the Java code written for this Stored Procedure.

```
import java.sql.*;
import org.apache.log4j.Logger;

/**
 * Simplified Stored Procedure example
 *
 */

public class SP1Reader {
	private static final Logger logger = Logger.getLogger(SP1Reader.class);
	private static final String db_connection = "jdbc:default:connection";
	private static boolean initialized = false;

	private static void reportSQLException(SQLException e) {
		// Unwraps the entire exception chain to show the real cause of the
		// Exception.
		while (e != null) {
			logger.error("  ----- SQLException -----");
			logger.error("  SQL State:  " + e.getSQLState());
			logger.error("  Error Code: " + e.getErrorCode());
			logger.error("  Message:    " + e.getMessage());
			e = e.getNextException();
		}
	}

	public static void Reader(String inString, java.sql.ResultSet[] result)
			throws Exception {
		/* This is the entry point of the Stored Procedure
		 * The entry point creates the connection, and calls the body of the SP
		 * passing the same attributes plus the connection
		 * object as the first attribute.
		 */
		/* The SP is declared in mxci as follows:

		create procedure SP1Reader( IN tableName varchar(255) )
		dynamic result sets 1
		external name 'SP1Reader.Reader(java.lang.String, java.sql.ResultSet[])'
		external path '/home/frans/projects/spj/spj.jar'
		reads SQL DATA
		language java
		parameter style java;
		*/

		try {
			if (!initialized) {
				logger.debug(System.getProperties());
				logger.info("Tracing JDBC to "
```

```
                                                + System.getProperty("jdbcmx.traceFile"));
                                logger.info("Connecting to " + db_connection);
                        }
                        Connection conn = DriverManager.getConnection(db_connection);
                        if (!initialized) {

                                logger.info("Driver version = "
                                                + conn.getMetaData().getDriverName() + " "
                                                + conn.getMetaData().getDriverVersion());
                                logger.debug("Connection catalog: " + conn.getCatalog());
                                logger.debug("UDR.catalog : "
                                                + System.getProperty("sqlmx.udr.catalog"));
                                logger.debug("UDR.schema  : "
                                                + System.getProperty("sqlmx.udr.schema"));
                                initialized = true;
                        }
                        Reader(conn, inString, result);
                } catch (SQLException e) {
                        reportSQLException(e);
                        throw e;

                }
        }

        public static void Reader(Connection conn, String inString,
                        java.sql.ResultSet[] result) throws Exception {
                String selectStmt = "Select * from zztables where table_name = ?";
        if (inString.equalsIgnoreCase("throwit") ) {
                  logger.info("Throwing a test user Exception");
                  throw new java.sql.SQLException("This exception is thrown upon request", "38088");
                }
                try {

                        logger.debug("Statement: " + selectStmt);

                        PreparedStatement ps = conn.prepareStatement(selectStmt);
                        ps.setString(1, inString);
                        result[0] = ps.executeQuery();
                } catch (SQLException e) {
                        reportSQLException(e);
                        throw e;

                } catch (Exception e) {
                        logger.fatal("Exception!!" + e.getMessage());

                }

        }

}
```

Learn more at
hp.com/go/NonStopDatabase

Sign up for updates
hp.com/go/getupdated

Share with colleagues

Rate this document

© Copyright 2013 Hewlett-Packard Development Company, L.P. The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

Oracle is a registered trademark of Oracle and/or its affiliates.

4AA4-9428ENW, October 2013